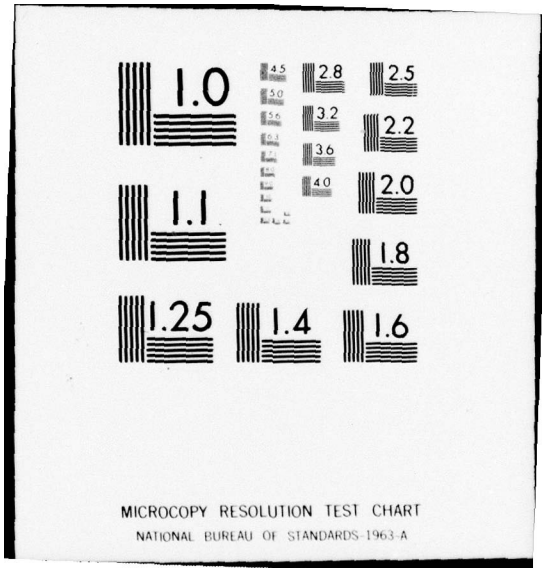
AD-A039 745 FEDERAL COBOL COMPILER TESTING SERVICE WASHINGTON D C F/G 9/2
STRUCTURED PROGRAMMING USING COBOL 74,(U)
MAY 77 P OLIVER, G N BAIRD
UNCLASSIFIED FCCTS/TR-77/09 NL

1 OF 1
ADA039 745

END

DATE
FILMED
6-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

STRUCTURED PROGRAMMING USING COBOL '74

Paul Oliver
George N. Baird
Department of the Navy
ADPE Selection Office
Software Development Division
Washington, D. C. 20376

ABSTRACT

Structural modularity in COBOL programming is achievable through variations of structured programming. The advantages of this approach as shown in a particular case study for a system reprogrammed using a structured approach. This paper also discusses the problems, approaches, techniques associated with the advancement, development, and implementation of structured programming using COBOL '74. Particular attention is given to micromodularity and to the costs, at the instruction level, of using a structured approach. Specific examples using several compilers are presented. Also examined will be those features in COBOL which are detrimental to structured programming and alternatives to using these features.

INTRODUCTION AND SCOPE

Gerald Weinberg, in his book The Psychology of Computer Programming (Ref. 1) suggests that we cannot really measure the goodness of programs on an absolute scale, and that we generally cannot even measure them on a relative scale. There is evidence that rapid quantification of software quality is not really feasible (Ref. 2), because simple formulas can often be misleading and hence not very credible.

Thus, we are led to less quantifiable measures of software quality, and a cursory glance at recent literature indicates that "simplicity" is a desirable characteristic. The simplification of a complex task can be achieved by modularizing it into separate, smaller tasks. We further require that each task be discrete and visible, that it be self-contained (thus constraining the assumptions it makes regarding the implementation of other tasks), that it have a single entry point and a single exit, and that when invoked by another task or module it returns to a standard point (Ref. 3).

One way of enhancing the simplicity of a program is through the concept of structured programming and the use of modularity, which is achieved through the separation, within a module, of data, processing code, and control, and through the maintenance of a simple, visible, control structure.

This paper concerns itself with how this can be accomplished using the COBOL language as defined in the 1974 COBOL Standard (Ref. 4). We will look at how structured constructs can be simulated using standard COBOL verbs, and discuss the micro-efficiency of using these constructs versus less disciplined techniques.

In an effort to take a realistic look at both the concept of structured programming and the tools available today in current programming languages, a pilot project was initiated by the Software Development Division of the Department of the Navy's Automatic Data Processing Equipment Selection Office. A program was selected from the Division's Programming Production Library. The project consisted of modifying the design of the program such that there was an increase in functional capability and the program could be compiled optionally taking advantage of the COBOL segmentation feature. This was not simply the recoding of an existing program using structured techniques. The basic design has to be oriented toward the concept of modularity and structured programming.

IMPLEMENTATION OF THE STRUCTURED PROGRAMMING CONSTRUCTS

Structured programming may be regarded as a set of rules and guidelines designed to enhance a program's readability and make the logic of the program more readily apparent. This results in the reduction of stylistic differences between programs written by different individuals thereby improving a programmer's ability to understand and modify existing programs. This also increases the programmer's ability to create and debug new programs. The guidelines used in the pilot project are as follows:

(a) Use of code formatting conventions, including indentation to represent control logic.

(b) Limit subroutine size (e.g., to the number of lines that can be contained on a single page).

(c) Limit the number of entry points and exit points to one each in both subroutines and subprograms.

(d) Limit logic control structure to the following basic structures.

(S = Statement, C = Condition)

- Simple sequence: S1 S2
- If-then-else: If C1 then S1 else S2.
- Do-while: While C1 do S1.

(e) Limit unconditional branches (GO TO) to reference only procedures within their own subroutine. (Avoid unconditional branches where possible.)

Unfortunately, there is not a one to one relationship between each of the basic structures mentioned above and language elements in COBOL 74. Therefore, for other than the simple sequence, a bit of simulation is required to produce the three basic constructs. The COBOL IF and PERFORM statements were used to accomplish this.

If-then-else

The if-then-else construct can be implemented in several different ways. The simplest and most direct approach is as follows:

```
IF condition
   statement-1
   statement-2
      .
      .
      .
   statement-n
ELSE
   statement-n+1
   statement-n+2
```

```
        .
        .
        .
    statement-n+m
```

The if-then-else *construct* must be terminated with a period to "end" the IF statement. If the condition is true then control passes to statement-1, statement-2 ...; at the completion of *statement-n* control is passed to the statement immediately following the IF statement, in this example it would be the statement immediately following statement-n+m. Should the condition be false, statement-1 through statement-n would be bypassed and control passed to statement-n+1.

If n and m are large and/or the coding being executed could be shared by other portions of the program then it would be to the programmers advantage to establish each set of instructions as a *subroutine* and reference them through the PERFORM statement.

```
IF condition
   PERFORM paragraph-1 THRU paragraph-1-exit
   ELSE
   PERFORM paragraph-2 THRU paragraph-2-exit.
```

There are inherent limitations in the definition of the COBOL IF/ELSE that can restrict the type of statement that can be contained in the *if-then-else* sequence. For example, if statement-1 is itself an IF statement then there cannot be any statement following statement-1 (statement-2 ...) which will be executed each time the outermost condition (condition-1) is true without regard to the outcome of the innermost condition (condition-2 making up statement-1).

```
IF condition-1
   IF condition-2
       statement-n
           .
           .
           .
   ELSE
       statement-n+1                    statement-1
           .
           .
           .
   statement-2
ELSE
   statement-3
```

There is no way to terminate the nested IF/ELSE statement until the ELSE related to the original IF statement is encountered. In the above case statement-2 would only be executed when condition-2 was false even though in the structure we have shown that, hierarchically, statement-1 and *statement*-2 are equivalent. *This* problem exists not only with the IF statement but any of the 17 COBOL statements which are considered conditional. (ADD with the ON SIZE ERROR *phrase*; CALL with the ON OVERFLOW phrase, etc.)

The problem of nesting conditions can be handled, as in the above example, by setting up statement-1 as a subroutine and reference it through the PERFORM statement.

```
IF condition-1
    PERFORM nested-condition
    statement-2
ELSE
    statement-3
        .
        .
        .
```

```
Nested-Condition
    IF condition-2
        statement-n
    ELSE
        statement-n+1
```

In this example, *statement*-2 will be executed each time condition-1 is true without regard to the outcome of the evaluation of the inner condition. There are basically two problems with conditional *statements* in COBOL 74 which, if corrected, could make structured programming easier.

1. The IF statement should be modified such that it resembled the DO ... END type of "vertical parenthesis" found in PL/1, this allows the statement between the DO and the END to be treated as a single syntactical unit by the compiler. The introduction of, for example, an "END-IF" that could terminate a nested IF statement would solve the problem. This would not be expected to be used in place of the traditional period but only in a situation where the period would not yield the desired results.

```
IF  condition-1
    statement-1
ELSE
    statement-2
END-IF
```

2. The other 16 conditional statements suffer from an additional problem; they are not consistent with the IF statement due to the lack of an ELSE option. The following ADD statement shows the inconsistency:

```
ADD identifier-1 TO identifier-2 ON SIZE
    ERROR imperative-statement.
```

The ON SIZE ERROR phrase represents a form of a conditional statement and the imperative-statement describes the actions to be taken should a size error condition exist. The imperative-sentence must be terminated by a period. This precludes the use of the above statement as well as 15 others from being used within the IF-THEN-ELSE construct.

```
IF condition-1
   ADD identifier-1 TO identifier-2
      ON SIZE ERROR imperative-statement.
```

The period following the imperative-statement terminates the IF as well as the imperative-statement. This precludes the use of the ELSE to specify what will happen only when condition-1 is false as well as the problem associated with the nesting of conditions described above.

The inclusion of an ELSE option as well as an END-IF statement would remedy these defects. For example, the above ADD format would read as:

```
ADD identifier-1 TO identifier-2
    ON SIZE ERROR
        statement-1
    ELSE
        statement-2
    END-IF
```

The above condition containing the ADD statement would then be coded as follows:

```
IF condition-1
    ADD A TO B ON SIZE ERROR
        statement-n
    ELSE                          statement-1
        statement-n+1
    END-IF
    statement-2
ELSE
    .
    .
    .
```

It should be noted that the restriction of having only an imperative statement in the above SIZE ERROR phrase would no longer be necessary.

Do-while

The DO WHILE is the basic looping structure and causes a set of procedures to be executed repeatedly as long as a particular condition is satisfied. The condition is tested prior to each execution of the set procedures (including the initial execution) until the condition is no longer true. At this point control passes to the next structure.

For COBOL, the PERFORM statement has a format that follows the DO WHILE definition closely, but not exactly. The PERFORM...UNTIL permits a set of procedures to be executed as long as a condition is not true and, as in the DO WHILE, tests the condition prior to each execution of the procedures including the first. The basic difference between the DO WHILE and the PERFORM...UNTIL is that the DO WHILE terminates when the condition is "true" and the UNTIL terminates when the condition is "false".

The implementation of the DO WHILE in COBOL is through the PERFORM statement and is accomplished simply by requiring negation of the condition being tested.

```
PERFORM paragraph THRU paragraph-exit UNTIL
    NOT condition-1.
```

This, in effect, says, continue to execute the referenced procedures until condition-1 is no longer true, or more precisely, continue to execute the procedures as long as the negated condition is true.

A counter/index/subscript can be initialized and incremented as the referenced procedures are being executed by including the VARYING phrase in the PERFORM statement.

```
PERFORM paragraph-1 THRU paragraph-1-exit
    VARYING identifier-1 FROM identifier-2
    BY identifier-3
    UNTIL NOT condition-1.
```

For the sake of convenience, two additional constructs were included in the projects programming procedures - DO UNTIL and CASE.

Do-until

The DO UNTIL is another structure which is used to accomplish basic looping. It is different from the DO WHILE in the respect that:

(1) The condition is tested after each execution of the referenced procedures (instead of before), so that the referenced procedures are always executed once even if the condition might already be true; and

(2) The test of the condition is reversed so that the referenced procedures are executed as long as the condition is not true. That is, the DO UNTIL terminates when the condition is true.

The COBOL implementation requires two statements for the DO UNTIL because the condition in the PERFORM ...UNTIL is always tested before the execution of the referenced procedures. This means that under the COBOL PERFORM statement the referenced procedures would not be executed if the condition is true at the time the PERFORM statement is executed. Therefore, the first execution of the procedures must be "forced".

```
PERFORM paragraph-1 THRU paragraph-1-exit.
PERFORM paragraph-1 THRU paragraph-1-exit
    UNTIL condition-1.
```

Case

The case construct is used to select one of a series of procedures to be executed, based on the interger value of an identifier. The contents of the identifier range from 1 up to and including the maximum number of procedures which are referenced. If the identifier contains a value less than 1 or greater than the maximum allowable, then no procedures are executed and control exits the case construct.

The COBOL implementation of the case construct requires the use of two verbs - the PERFORM and the GO TO ... DEPENDING.

```
PERFORM case-paragraph THRU case-exit
    .
    .
    .
case-paragraph.
    GO TO case-1, case-2, ..., case-n
        DEPENDING ON identifier.

    GO TO case-exit.

case-1.
    (code for case-1)

    GO TO case-exit.

case-2.
    (code for case-2)

    GO TO case-exit.
    .
    .
    .
    .
case-n
    (code for case-n)

    GO TO case-exit.

case-exit.
    EXIT.
```

An alternative method of implementing the case construct when the number of procedures is small could be accomplished through the use of the IF and PERFORM statement.

```
IF case-identifier EQUAL TO 1 PERFORM case-1.
```

IF case-identifier EQUAL TO 2 PERFORM case-2.
.
.
.
IF case-identifier EQUAL TO n PERFORM case-n.

Efficiency consideration

The use of the COBOL implemented (structured programming) constructs described above, based on our experience, is not inefficient. It has been suggested that the inclusion of the PERFORM statements necessary to support structured programming coupled with the elimination of the GO TO statement (where possible) would increase program memory and time requirements.

The evaluation of the type of object code generated by the structured program constructs shows that for the most part it is no worse than that produced by a programmer avoiding structuring. Keep in mind that the increase of the visibility of the program logic and degree to which debugging and maintenance become easier can far outweigh a small increase of memory or time utilization.

- PERFORM vs GO TO

  The PERFORMing of a subroutine requires the execution of 5 machine instructions under the IBM System 360 OS/V4 COBOL compiler. The use of two GO TO statements (one to get there and one to get back) would require 2 machine instructions and preclude the use of that set of code by any other part of the program. The use of the ALTER verb could modify the return GO TO such that the subroutine could be shared by other parts of the program but the overhead would add 2 machine instructions and further cloud the logic flow of the program. In the case of the if-then-else, the GO TO could not be used in lieu of the PERFORM statement to reference a subroutine.

- DO UNTIL

  The PERFORM statement in this case produces 18 machine instructions and is controlling the execution of the loop by varying a counter until the referenced procedures have been executed a specific number of times. Consider the alternative of the programmer using the GO TO and controlling the execution of the loop himself

  - Set lower and upper bounds
  - Set the counter
  - Increment the counter
  - Test condition and branch

  These require 14 machine instructions on the same compiler. The reduction of 4 machine instructions hardly warrants the loss of control logic visibility and the requirement of the programmer to code 6 additional COBOL statements, two of which would be GO TO statements.

- DO WHILE

  The coding generated by the HIS 6000 COBOL Compiler (Version WW5.1 EIS) for the PERFORM statement in this case produces 15 machine instructions and is causing the execution of the loop until a condition is no longer satisfied. The execution of the same set of procedures using the GO TO and an IF statement to determine whether to continue executing the procedure causes 14 machine instructions to be generated. At this point it becomes questionable as to whether the use of GO TO should ever be considered in light of the saving of only 1 machine instruction from 15 to 14.

Project results

The previously mentioned project included the rewriting of a COBOL program in the Programming Production Library which is used for library maintenance of source programs and their related data. The statistics on the original program were:

Source statements	Data Division	471
Source statements	Procedure Division	2,001
Core Required		21,953 words
Possibilities of Segmenting the Program		NO

The new program was basically a redesign of the old one, in which no functional capability was lost and additional capabilities were added has the following statistics:

Source statements	Data Division	645
Source statements	Procedure Division	2,145
Core Required		22,115 words
Possibilities of Segmentation		Yes/14,211 words

It is only fair to point out that the increase in source program statements may be misleading since the formatting used could require more source lines. A fairer comparison might be the number of tokens in the Procedure and Data Division.

The previous program was developed over a period of several years and was periodically modified to handle new requirements. Its modularity had been aborted and at the time of its retirement still had problems which were left uncorrected. The redesign took 15 man hours. Programming was accomplished in 72 man hours (by several programmers). Testing was accomplished as the various modules were coded (top down design was used).

The integration and testing (syntax checking not counted) of the program was completed after five test compilations. There were three serious errors involved in the testing phase. Two of these errors were found to be caused by a "GO TO" statement.

Much like the upkeep of an older automobile the expense of repairing the old program soon began to outweigh the cost of creating the new one. The statistics for this project will not be complete until the new program has been under maintenance for a period of time. At that point a review of the effort required to make changes to the program in light of its structured/modular design will be necessary. Thus far we are pleased with the apparent results.

CONCLUSION

Our conclusions are that there is an obvious advantage to using structured programming. The pilot project using COBOL as well as others indicate this is true. Although we were able to use COBOL adequately it does require the programmer to accomplish his task in spite of the current language constructs.

The modifications necessary to COBOL to make it more amenable to structured programming are as follows:

1. IF-THEN-ELSE.

   The ability must be provided to terminate a condition without having to use a period or in some cases an ELSE related to a superior condition. This could be done with an END-IF or some other construct as shown under the discussion of the IF-THEN-ELSE.

2. DO WHILE

   An additional option in the PERFORM statement which might read as 'PERFORM ... WHILE condition' would eliminate the need for the programmer to think negatively (i.e. PERFORM ... UNTIL NOT condition). This would amount to a relatively simple modification to the COBOL language and compilers.

3. DO UNTIL

   The philosophy of the DO UNTIL is logically backwards relative to the PERFORM UNTIL. This is because the PERFORM UNTIL checks the condition prior to the execution of the referenced procedure and if true the first time the procedures are to be executed, the procedures will not be executed at all. The DO UNTIL checks the condition after the execution of the referenced procedure and always insures that at least one execution will take place even when the condition is true before the DO UNTIL is executed. This is probably a critical problem in relation to 1 and 2 above because it directly effects overhead in core requirements due to the requirement of back to back PERFORM statements to accomplish this feat. A modification to the semantic action of the PERFORM UNTIL would be an unmitigated disaster. This could wipe out existing program which had been running correctly for years. It appears that a new construct may have to be created to accomplish the DO UNTIL perhaps even a DO UNTIL statement.

4. CASE

   Although the GO TO DEPENDING is bulky and requires the use of GO TO statements to exit the case procedure it is adequate until a better idea comes along.

We feel the appropriate modifications should be made to COBOL as soon as possible so that implementation experience gained can be such that they can be modified such that they are ready for inclusion in the next COBOL standard.

REFERENCES

1. Weinberg, G. M., The Psychology of Computer Programming, Van Norstrand Reinhold, 1971.

2. Boehm, B. W., et al, "Characteristics of software quality", TRW-SS-73-09, December, 1973.

3. Armstrong, R. M., Modular Programming in COBOL, John Wiley and Sons, 1973.

4. American National Standard Programming Language COBOL, X3.23-1974, American National Standards Institute, Inc., 1974.

BIBLIOGRAPHIC DATA SHEET	1. Report No. FCCTS/TR-77/09	2.	3. Recipient's Accession No.
4. Title and Subtitle Structured Programming Using COBOL 74			5. Report Date 9 May 1977 6.
7. Author(s) Paul Oliver and George N. Baird			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Software Development Division ADPE Selection Office Department of the Navy Washington, D. C. 20376			10. Project/Task/Work Unit No. 11. Contract/Grant No.
12. Sponsoring Organization Name and Address ADPE Selection Office Department of the Navy Washington, D. C. 20376			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

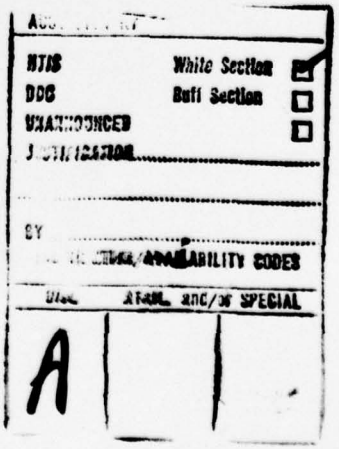
Structural modularity in COBOL programming is achievable through variations of structured programming. The advantages of this approach as shown in a particular case study for a system reprogrammed using a structured approach. This paper also discusses the problems, approaches, techniques associated with the advancement, development, and implementation of structured programming using COBOL '74. Particular attention is given to micromodularity and to the costs, at the instruction level, of using a structured approach. Specific examples using several compilers are presented. Also examined will be those features in COBOL which are detrimental to structured programming and alternatives to using these features.

17. Key Words and Document Analysis. 17a. Descriptors

Structured Programming
COBOL 74

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group 09/02



18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited.	UNCLASSIFIED	5
	20. Security Class (This Page) UNCLASSIFIED	22. Price